

**BEST AVAILABLE COPY**

3. The undersigned Mark Linus Bauman, Bob Richard Cernohous, Kent L. Hofer, John Charles Kasperski, Steven John Simonson, and Jay Robert Weeks, each declares on his own behalf that all statements made herein of their own knowledge are true, and that all statements made upon information and belief are believed to be true. Further, that statements made herein are made with the knowledge that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. § 1001, and that any willful false statements may jeopardize the enforceability of the above-entitled patent application, or any patents issued thereon.

5/24/2005  
Date

Mark Linus Bauman  
Mark Linus Bauman

5/23/05  
Date

Bob Richard Cernohous  
Bob Richard Cernohous

5/26/05  
Date

Kent L. Hofer  
Kent L. Hofer

5/24/05  
Date


John Charles Kasperski  
John Charles Kasperski

5/23/05  
Date

Steven John Simonson  
Steven John Simonson

5/19/05  
Date

Jay Robert Weeks  
Jay Robert Weeks

	<div data-bbox="235 777 316 1533" style="background-color: black; height: 360px; width: 50px;"></div> <div data-bbox="341 724 438 1533" style="background-color: black; height: 385px; width: 60px;"></div> <div data-bbox="341 798 389 1533" style="position: absolute; left: 210px; top: 380px;">Created</div> <div data-bbox="349 798 389 1186" style="position: absolute; left: 215px; top: 380px;">On 04/10/2000 03:29:44 PM EDT</div>
---	---

Required fields are marked with the asterisk (\*) and must be filled in to complete the form .

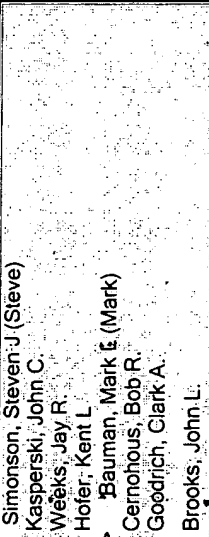
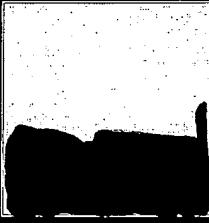
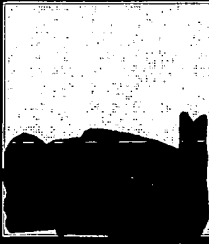

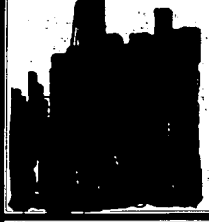
\*Title of disclosure (in English)

Method for Determining on Demand Right Size Buffering Within a Socket Server Implementation

<input type="checkbox"/> Status	<input type="checkbox"/> Final Decision <input type="checkbox"/> (File)
---------------------------------	---

EXHIBIT A



<p>Simonson, Steven J (Steve)  Kasperski, John C  Weeks, Jay R  Hofer, Kent L  &gt; Bauman, Mark E (Mark)  Cernohous, Bob R  Goodrich, Clark A  Brooks, John L</p>					
--	---	---	--	---	---

> denotes primary contact

To set the IDT Team for a specific Functional Area and/or Technology Code, click on "Select...Functional Area" or "Select...TechnologyCode" in the action bar.

Attorney/Patent professional

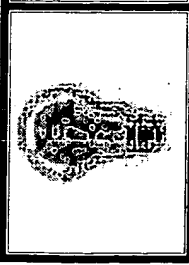
IDT team



[REDACTED]

[REDACTED]

[REDACTED]



[REDACTED]

Archived On 04/14/2001 12:02:53 AM

### Title of disclosure (in English)

Method for determining on demand right size buffering within a socket server implementation.

1. Describe your invention, stating the problem solved (if appropriate), and indicating the advantages of using the invention.  
This invention could drastically reduce server storage requirements necessary to receive inbound client requests.

This invention is relevant to synchronous and asynchronous socket receive APIs.

Currently sockets receive semantics are either synchronous or asynchronous. Synchronous APIs such as `recv()` and `recvmsg()`, receive data in the execution context issuing the API. Asynchronous APIs such as `asyncRecv()` return indications that the receive will be handled asynchronously if the data is not immediately available. When the data arrives the supplied buffer is filled and the completed request transitions to a common wait point for processing.

Synchronous receive I/O will wait until the data requested arrives. This wait is typically performed within the sockets level of the operating system. During this wait the buffer supplied by the application server is reserved until this receive completes successfully or an error condition is encountered. Many client connections have a bursty data nature where there can be significant lag times between each client request. The buffers reserved for the incoming client requests can typically sit idle awaiting for client requests to be received. This can cause additional storage to be

allocated but not used until the data arrives. Memory is a limited resource. When there are multiple allocated buffers that are underutilized, system paging rates can be adversely affected.

Asynchronous I/O registers a buffer to be filled asynchronously when the data arrives. This buffer cannot be used until the I/O completes or an error condition causes the operation to fail. When data arrives this buffer is filled asynchronously to the server process. This asynchronous behavior is advantageous, but again the buffer supplied is reserved until the operation completes and indication is returned to the application server. Storage and paging concerns described for synchronous receive I/O above also apply for this asynchronous I/O processing.

There are two issues associated with above processing:

- 1) Multiple buffers reserved at any given time are more than what are needed to service the number of incoming requests. This means that the memory footprint for processing is much larger than needed to service client connections.
- 2) Memory allocated for each incoming request will consume this valuable resource and cause memory management page thrashing.

Ideally it would be best to acquire a buffer large enough to hold all of the data when it arrives. This is referred to as "on demand right size buffering". The buffer is acquired when the data arrives and is large enough to contain all of the data. This behavior would best utilize buffers and keep the buffer highly utilized from a memory management paging perspective.

One potential problem involves determining what size buffer the application server should provide when the I/O operation is initiated. The record length is contained within the input datastream and will only be known when the data arrives. The application server could be coded for worst case and always supply a buffer large enough to accommodate the largest record possible. This would be a waste of resources and could adversely affect the paging rates for not only the server but the system itself.

[REDACTED] This allows the operating system to bundle I/O on record boundaries and reduce CPU processing.

This invention resolves the problems stated above by defining a method for the operating system to obtain a right size buffer when the client data arrives.

2. How does the invention solve the problem or achieve an advantage, (a description of "the invention", including figures inline as appropriate)? This invention defines three buffering modes which can be used to efficiently allocate and right size data for both synchronous and asynchronous receives. These buffering modes are flexible enough to meet a wide range of application server buffer demands.

First, to address asynchronous I/O where the buffer is reserved until the data arrives and the request is handled asynchronously to the server

application.

Enhance asynchronous processing by adding a buffer mode value to the asynchronous receive API. For example `asyncRecv()` could be enhanced with a `buffer_mode` parameter. The `buffer_mode` parameter could have three possible values:

- `caller_supplied`
- `caller_supplied_dynamic`
- `system_supplied`

#### `caller_supplied`

Basically this is the common receive buffer methodology used today. The caller supplies a `buffer` (pointer to) and `buffer_length` on the API call. The buffer cannot be used until the operation completes and indication of completion has been received by the server application. This may not be optimal if there are thousands of bursty client connections. Buffer pages may be under utilized thereby affecting system paging rates. The application cannot use the buffer until the operation is complete. The operating system uses the buffer to load the data buffer asynchronously to the server application.

#### `caller_supplied_dynamic`

Allows the application server to supply a function to be called by the operating system in order to obtain a right-sized buffer. This buffer comes from server owned storage. No buffer pointer needs to be supplied on the `asyncRecv()` call, but `buffer_length` is. Here `buffer_length` represents the amount of data requested. The server supplied program is called when the completed entry is consumed from the common async completion wait point. At this instant the operating system would determine the buffer size required and:

- call the server supplied function to acquire a right-sized-buffer
- copy the data into that buffer
- return the completed operation to the server.

This mode allows the application to use it's own storage allocation scheme whatever that may be.

Typically the data copy here would not happen asynchronous to the server thread. For some server implementations this may not be advantageous when running on a multi processor system. To provide for asynchronous copies, `caller_supplied_dynamic` mode will allow the server to optionally supply a buffer to be used. If the supplied buffer is large enough then it will be used and the data copies will be done asynchronously. If the supplied buffer is not large enough then another buffer will be acquired by the call back function mentioned above. The buffer call back function interface coded in C could be as follows:

```
typedef char * (*get_buffer_call_back_t) (int*, char*);
```

The function takes as input:



- char \* -- If not NULL then this points to the application buffer that was too small to service the receive request. The operating system is returning this buffer to the application.

The function takes as input and output:

- int \* -- size of buffer needed on input and the size of buffer returned on output

Returned as output:

- char \* -- pointer to requested buffer.

In summary this buffer\_mode supports:

- Buffers can be acquired when data arrives and is ready to be consumed. Buffer space is highly utilized.

[REDACTED]

- Allows the application server to plug in it's own data buffering scheme and still get the documented advantages.
- Optionally, the application server can supply a buffer which will be used if it can be. If not, then a mechanism is defined to return this buffer and acquire a right size buffer.

#### system\_supplied

In this mode the system could allocate and manage buffer resource for an application server. The application need not specify a buffer, only `buffer_length` which represents the amount of data requested. If the application does supply a buffer, it must be a `system_supplied` buffer that will implicitly be freed or cached. (This eliminates the overhead of a separate call to `free_buffer()`). The operating system would acquire process scoped storage (heap based) when the data arrives. The buffer acquisition can occur in a task which will assure all data copies happen asynchronous to the server process.

[REDACTED]

Typically the data copy here would happen asynchronous to the server thread. When the server obtains and finishes processing the completed request, the buffer is returned to the system for reuse. The system will keep a cache of these buffers to limit the heap allocation and deallocation costs. In order to limit the number of buffers and in turn heap fragmentation, the smallest buffer will be 128 bytes and increase in size by a power of 2, to a reasonable size. The system shall define the interfaces for the server to acquire and free these cached system buffers. For example:

```
void * get_buffer(int* sizeOfBuffer)
```

Input

The minimum desired size of the buffer.

Output

System supplied buffer

```
int free_buffer (void * bufferToFree)
```

Input

Buffer to free  
Output

Indication if buffer was freed.

Note: The system may optionally not deallocate the buffer but instead cache the buffer for reuse.

In summary this `buffer_mode` supports:

- Buffers can be acquired when data arrives and is ready to be consumed. Buffer space is highly and efficiently utilized.
- Data copies are asynchronous to the server process's processing.
- System will keep a buffer cache to cut down on allocation and deallocation costs.
- Allows application to acquire and free buffers from this cache.

These additional options to the `asynRecv()` along with the additional APIs described above, will allow the application server to manage its own buffer space, manage parts of it, or hand the total management effort over to operating system. This buffer methodology will minimize paging rates and also support record based I/O.

Also, synchronous APIs such as `recvmsg()` and `readv()` can also take advantage of this methodology. A new `ioctl()` command definition could be defined which would register the type of buffer mode to be used for any particular socket (client connection). This along with the get and free APIs can supply the same advantageous for synchronous receives.

The following diagram flow shows the basic `caller_supplied` buffer mode. Data is copied asynchronously. This support is available today. Problems associated with this type of support are:

- data buffer for incoming data requests is allocated and reserved until the data arrives and is consumed by the application.
- while waiting for the data, the buffer storage might be paged out of memory, thus incurring the cost of paging the storage back into memory when the data is loaded into the buffer.
- the data buffer supplied is unlikely to be the optimal size for the data



asyncRecv with caller\_supplied buffer\_mode and the buffer is large enough to contain the largest possible record.



asyncWait

sleep



a communications router task

No data arrives for a substantial period of time. The arriving data is copied to the user buffer as it arrives. The record only partially fill the user's buffer.

wakeup asyncWait

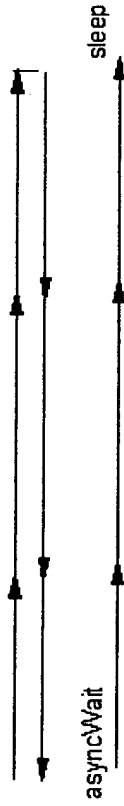
The application has received the client request in caller\_supplied storage. But more storage than necessary was allocated for the request. The extra was held longer than necessary.

The following diagram flow shows the basic caller\_supplied\_dynamic buffer mode. This support is not available today and is part of the advantages of this invention. Advantages of this type of support are:

- data buffer for incoming data is obtained at the time it is needed, paging rate is minimal
- data buffer is correctly sized based on the data request, and thus the storage is efficiently and fully utilized
- [REDACTED]
- this implementation facilitates plugging into the buffering allocation model of the application



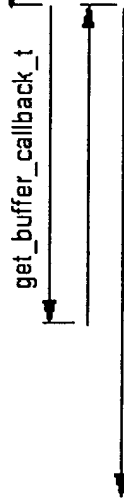
asyncRecv with caller\_supplied\_dynamic  
buffer\_mode and no buffer is specified.



a communications router task

The arriving data will be queued internally until it has all arrived (as defined by the original requested length or the record definition) and the user callback will be called to get a right-sized buffer.

get\_buffer\_callback\_t



The right-sized buffer has been acquired from the application server. Data is copied to the right-sized buffer and returned.

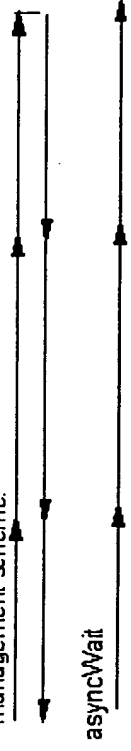
The application has received the client request in caller\_supplied\_dynamic storage. When it is done processing the request, it manages the storage with it's own memory management scheme.

The following diagram flow shows another aspect of the caller\_supplied\_dynamic buffer mode. This support is not available today and is part of the advantages of this invention. Advantages of this type of support are:

- [REDACTED]
- data is copied asynchronously into the buffer [REDACTED]
- this implementation facilitates plugging into the buffering allocation model of the application



asyncRecv with caller\_supplied\_dynamic buffer\_mode and a "typical sized" buffer is specified from the application's memory management scheme.



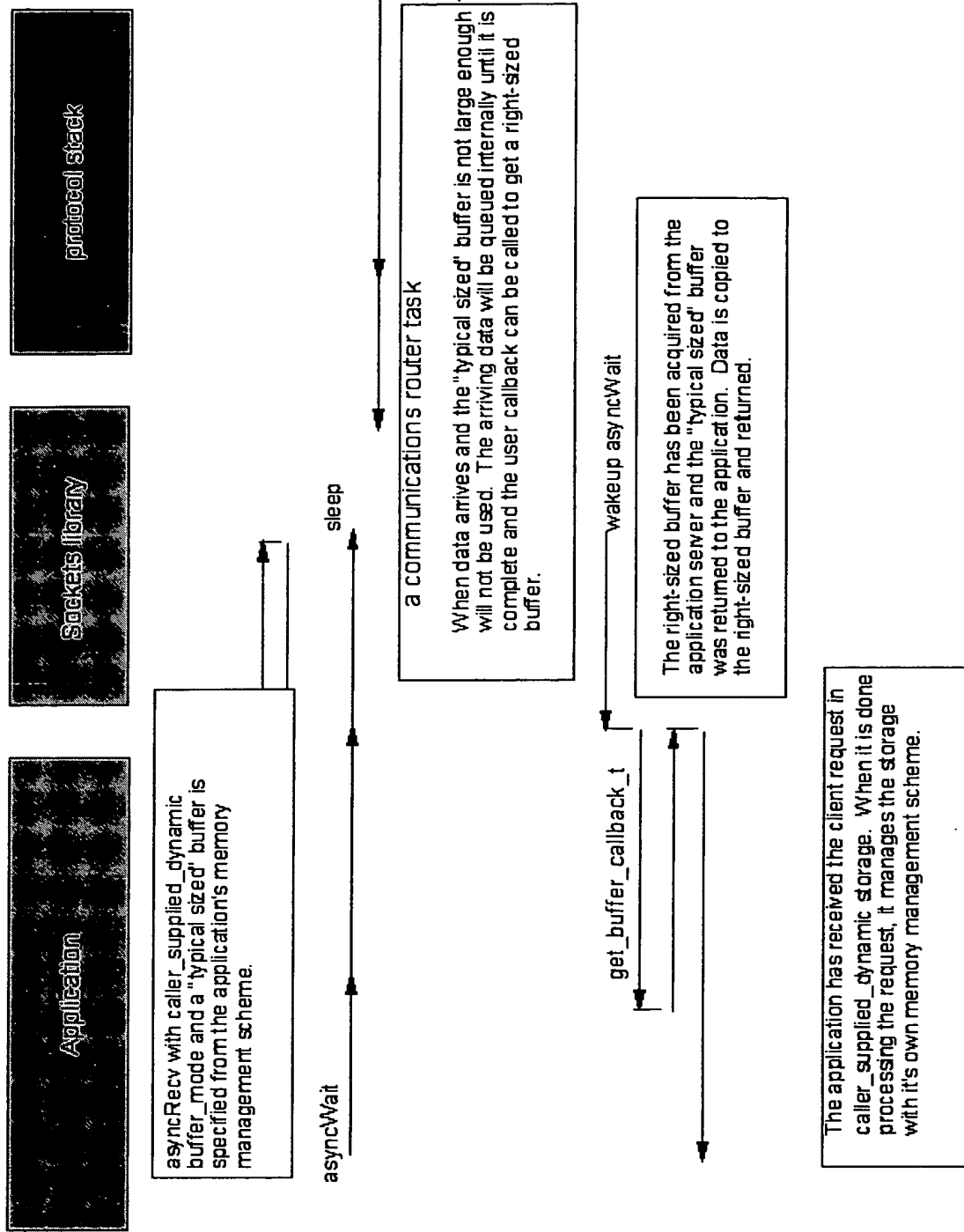
a communications router task

When a full record arrives and the "typical sized" buffer is large enough, the record will be asynchronously copied into the user's buffer.

The application has received the client request in caller\_supplied\_dynamic storage. When it is done processing the request, it manages the storage with it's own memory management scheme.

The following diagram flow shows yet another aspect of the `caller_supplied_dynamic_buffer` mode. This support is not available today and is part of the advantages of this invention. This flow is the same as the previous except that it will obtain a right sized buffer if the supplied buffer is not large enough to contain the data.

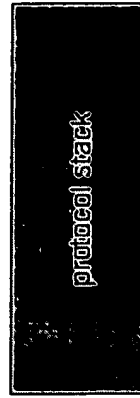




The following diagram flow shows the basic system\_supplied buffer mode. This support is not available today and is part of the advantages of this invention. system\_supplied buffer mode has the most advantages of all of the buffer modes discussed as part of this invention disclosure. The buffer is allocated "on demand" and is "right sized". The data is copied asynchronously.

Advantages of this type of support are:

- data buffer for incoming data is obtained at the time it is needed, paging rate is minimal
- data buffer is correctly sized based on the data request, and thus the storage is efficiently and fully utilized
- automatic buffer allocation and caching is enabled and managed by the system (which should improve performance of this flow compared to previous flows)



asyncRecv with system supplied buffer mode



a communications router task

When data arrives, system supplied storage is allocated or taken from a cache. The length is based on the original asyncRecv request or on the record definition. If it was a record definition, sockets waits until the entire record arrives and right-sizes the allocated storage.

wakeup asyncWait

The application has received the client request in system-supplied storage. When it is done processing the request, it can release the storage with free\_buffer() or implicitly free the buffer by using it on the next asyncRecv.

asyncRecv with system supplied buffer\_mode



the previous buffer is freed

a communications router task

When data arrives, system supplied storage is allocated or taken from a cache. The length is based on the original asyncRecv request or on the record definition. If it was a record definition, sockets waits until the entire record arrives and right-sizes the allocated storage.

wakeup asyncWait

3. If the same advantage or problem has been identified by others (inside/outside IBM), how have those others solved it and does your solution differ and why is it better?

This invention is a unique combination of socket API changes, socket server design changes, system interface changes and storage management which can substantially improve buffer management and reduce storage requirements of socket server applications.

FURTHER MATERIAL REDACTED

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

**BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☒ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**